EECS 322 Computer Architecture

The Multi-Cycle Processor

Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

Assembling Branch Instructions (chapter 3)

Branch beq $rs,$rt,offset16

$pc = ($rt == $rs)? ($pc+4+(δ^{32}(offset16)<<2))):($pc+4);

Suppose the fib_exit = 0x81fc084C, pc = 0x81fc08124,

beq $s3,$s7,fib_exit

Relative addr = addr–(pc+4) =0x81fc084C–0x81fc08128 =0x24

Then rel addr>>2 = fib_exit >> 2 = 0x00000024 >> 2

= 0000 0000 0000 0000 0000 0000 0010 0100>>2

= 0000 0000 0000 0000 0000 0000 0000 1001

= 0x0000009

0x1fc08124 beq $s3,$s7,fib_exit

000100: 00011 00111 0000000000001001

Executing Branch Instructions

Branch beq $rs,$rt,offset16

$pc = ($rt == $rs)? ($pc+4+(δ^{32}(offset16)<<2))):($pc+4);

Suppose the pc=0x81fc08124,

beq $s3,$s7,fib_exit

000100: 00011 00111 0000000000001001

Then address =0x00000009

Then address << 2 = 0x00000024

Then $pc+4 = 0x81fc08128

Then $pc+4 + address<<2 = 0x81fc0814c

If branch occurred then pc = 0x81fc0814c

else pc = 0x81fc08128

Signed Binary numbers (chapter 4)

Assume the word size is 4 bits,

Then each bit represents a power = $[-2^3]2^2 2^1 2^0$ = S421

S represents the minus sign bit = $-2^3 = -8$

S421	
0000	0
0001	1
0010	2
0011	3 = 2+1
0100	4
0101	5 = 4+1
0110	6 = 4+2
0111	7 = 4+2+1

S421	
1000	–8 = –8+0
1001	–7 = –8+1
1010	–6 = –8+2
1011	–5 = –8+2+1
1100	–4 = –8+4
1101	–3 = –8+4+1
1110	–2 = –8+4+2
1111	–1 = –8+4+2+1

unsigned 4 bit number: 0 to 2^4 = 0..15

signed 4 bit number: -2^3 to $2^3 - 1$ = –8 .. 7

Sign numbers causes the loss of 1 bit accuracy

This is why C language provides signed & unsigned keywords

1's and 2's complement

One`s complement: invert each bit

For example: 0100 becomes 1011 (Note: 1011 is –5 and not –4)

The C language has a 1's complement bitwise operator tilde (~).
(i.e. ~1011 becomes 0100)

The 1's complement operator has the property: X = ~~X;

Two's complement number (also negation) is expressed as

two's complement = –X = (~X)+1

The 2's complement operator has the property: X = – –X;

For example: 4 becomes –4

For example: 0100 becomes (1011+0001) = 1100 = –4

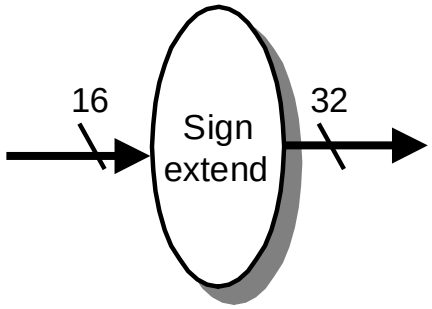
Sign extension

Suppose we want to sign extend 4 bit word to 8 bits

Then take the value of the sign bit and propagate it

For example: 1011 becomes 11111011

- Two's complement allows the number to retain the same value even though we are adding 1's!
- 11111011 = –128 + 64 + 32 + 16 + 8 + 2 + 1 = –5
- 1011 = –8 + 2 + 1 = –5
- Two's complement allows us to treat the sign bit as another digit!



1-bit addition

The rules to add 1 bit numbers: Cin=Carry in; Cout=Carry Out

Input			2^1	2^0	
A	B	Cin	Cout	Sum	$=2^1\ 2^0$
0	0	0	0	0	=0
0	0	1	0	1	=1
0	1	0	0	1	=1
0	1	1	1	0	=2
1	0	0	0	1	=1
1	0	1	1	0	=2
1	1	0	1	0	=2
1	1	1	1	1	=3

Sum = oddparity(A, B, Cin) = “odd number of bits”

Cout = majority(A, B, Cin) = “majority vote”

1-bit sum is the same as adding the bits modular 2 (i.e base 2).

N-bit addition

N-bit addition requires using only the 1-bit addition table

Suppose we want to add: 0101 + 0011 = 5+3 = 8

Cin	1	1	1	
5	0	1	0	1
3	0	0	1	1
Sum	1	0	0	0
Cout	0	1	1	1
Overflow 1				

If the word size is a 4 bits then the Sum of 1000 is really -8

which is incorrect. Hence the number field is too small.

This is called arithmetic overflow = Cin_{sign} ^ $Cout_{sign}$

Is the exclusive-or of the Cin and the Cout of the sign bit field

N-bit subtraction

Two's complement allows us to treat

N-bit subtraction as N-bit addition.

Suppose we want to add: 5 – 3 = 0101 – 0011 = 3

First 2's complement 3: 0011 ⇒ 1100+1 ⇒ 1101

Now just do addition: 5 + –3 = 0101 + 1101

Cin	1		1		
5	0	1	0	1	
–3	1	1	0	1	
Sum	0	0	1	0	=2
Cout	1	1	0	1	
Overflow=0					

arithmetic overflow bit = Cin_{sign} ^ $Cout_{sign}$ = 1 ^ 1 = 0

Multiply instruction

Two's complement allows us to also multiply by addition

$1 \times 1 = 1$ and $0 \times M = 0$

Warning: for each sub-product, you must extend the sign bit

Note: a $N \times N$ multipy results in a 2N product = $4 \times 4 = 8$ bit

–3	1	1	1	1	1	1	1	0	1
5						0	1	0	1
	1	1	1	1	1	1	1	0	1

1	1	1	1	1	1	0	1		
1	1	1	1	1	1	0	0	0	1

Thus a 4×4 multiply = 8 bit product. Add time = 1 clock.

N x N Multiply

Easier to place positive on top?

5						0	1	0	1
–3		1	1	1	1	1	1	0	1
						0	1	0	1
				0	1	0	1		
			0	1	0	1			
		0	1	0	1				
	0	1	0	1					
0	1	0	1						
1	0	1							
1	1	1	1	1	1	0	0	0	1

Add time = 6 clocks

MIPS multiply instruction

The MIPS does not have a general purpose multiply.

It is required to copy the values in to special registers.

Also, 32 x 32 multiply results in a 64 bit product.

In fact, some RISC machines, use only shift instructions and claim the same performance as machines with a hardware multiply!

Unsigned multiply:	multu	$rs,$rt	# hi_lo_64 = $rs * $rt
Signed multiply:	mult	$rs,$rt	# hi_lo_64 = $rs * $rt
move from low:	mflo	$rd	# $rd = lo
move from high:	mfhi	$rd	# $rd = hi

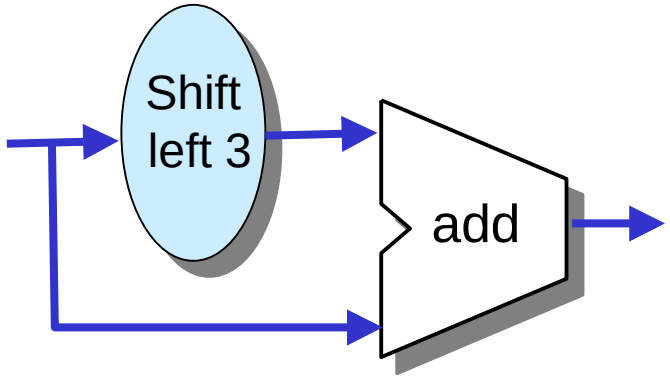
What is the MIPS for the following C code?

```
int x, y;        y = 9*x + 2;
```

Multiply by powers of 2: using shift

- The binary radix allows for easy multiply by powers of 2.
- The reason to use shifting is because it is fast (just move bits).
- In fact, many computers use a barrel shifter.
- A barrel shifter, shifts any amount in one clock cycle.
- Whereas a so-so multiplier my take up to n clocks.
- Multiply by a constant allows for further optimization
- For example, x*9 = x*(8+1) = x*8 + x*1

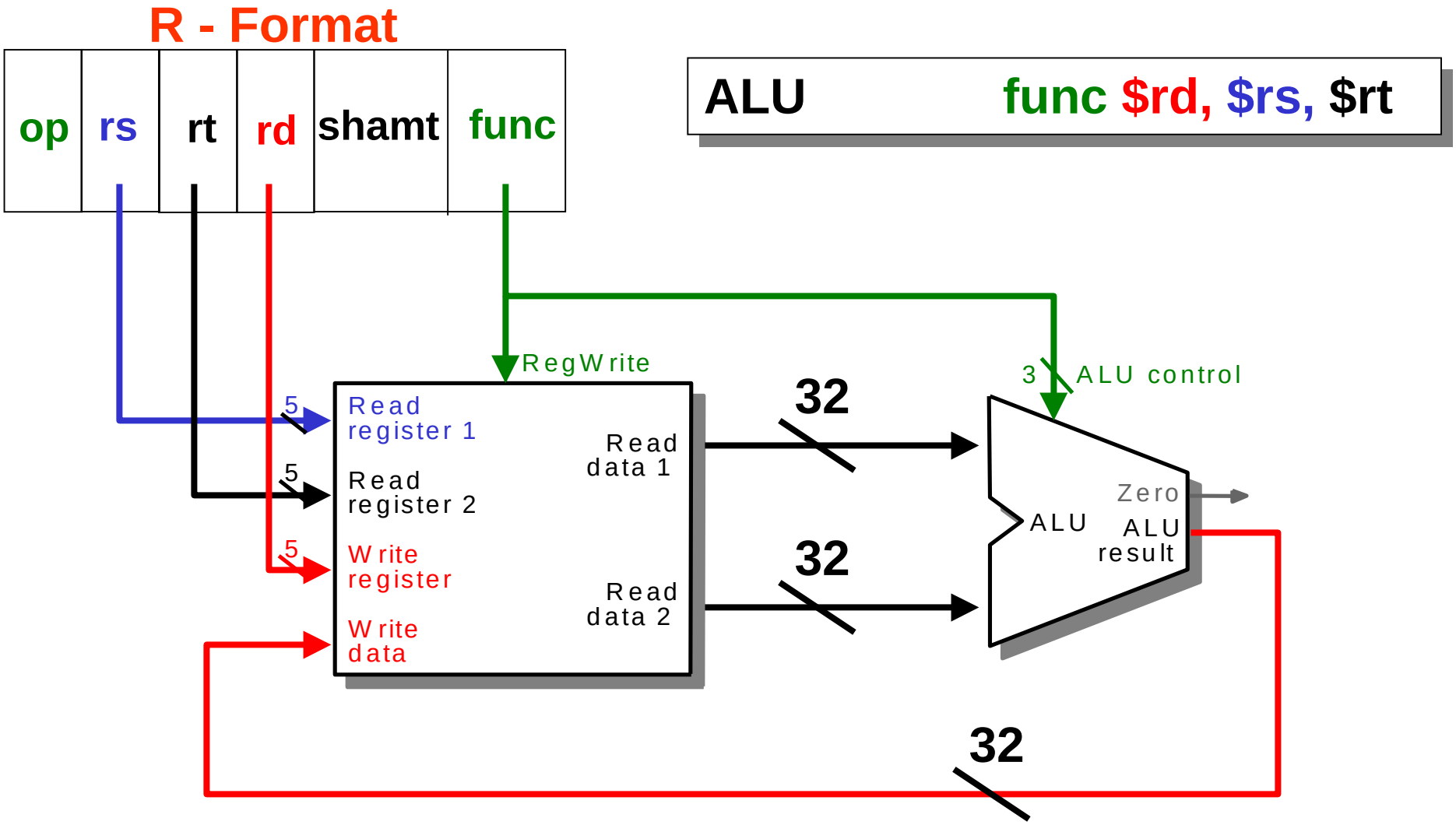
```
sll     $s1,$s0,3          # 8 = 2^3
add     $s1,$s0,$0         # x*9
```



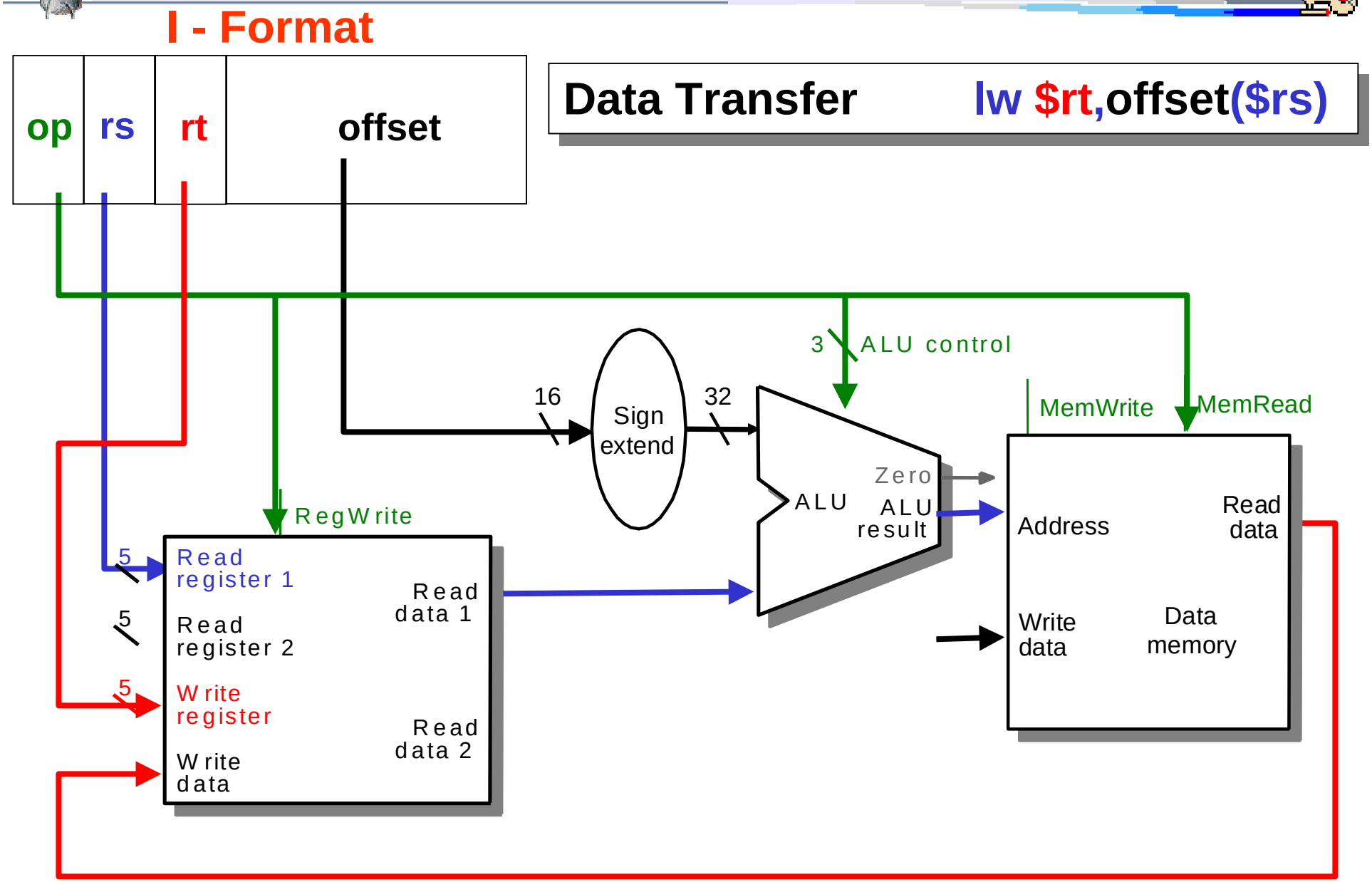
What is the MIPS for the following C code?

```
int x, y;        y = 18*x + x/4;
```

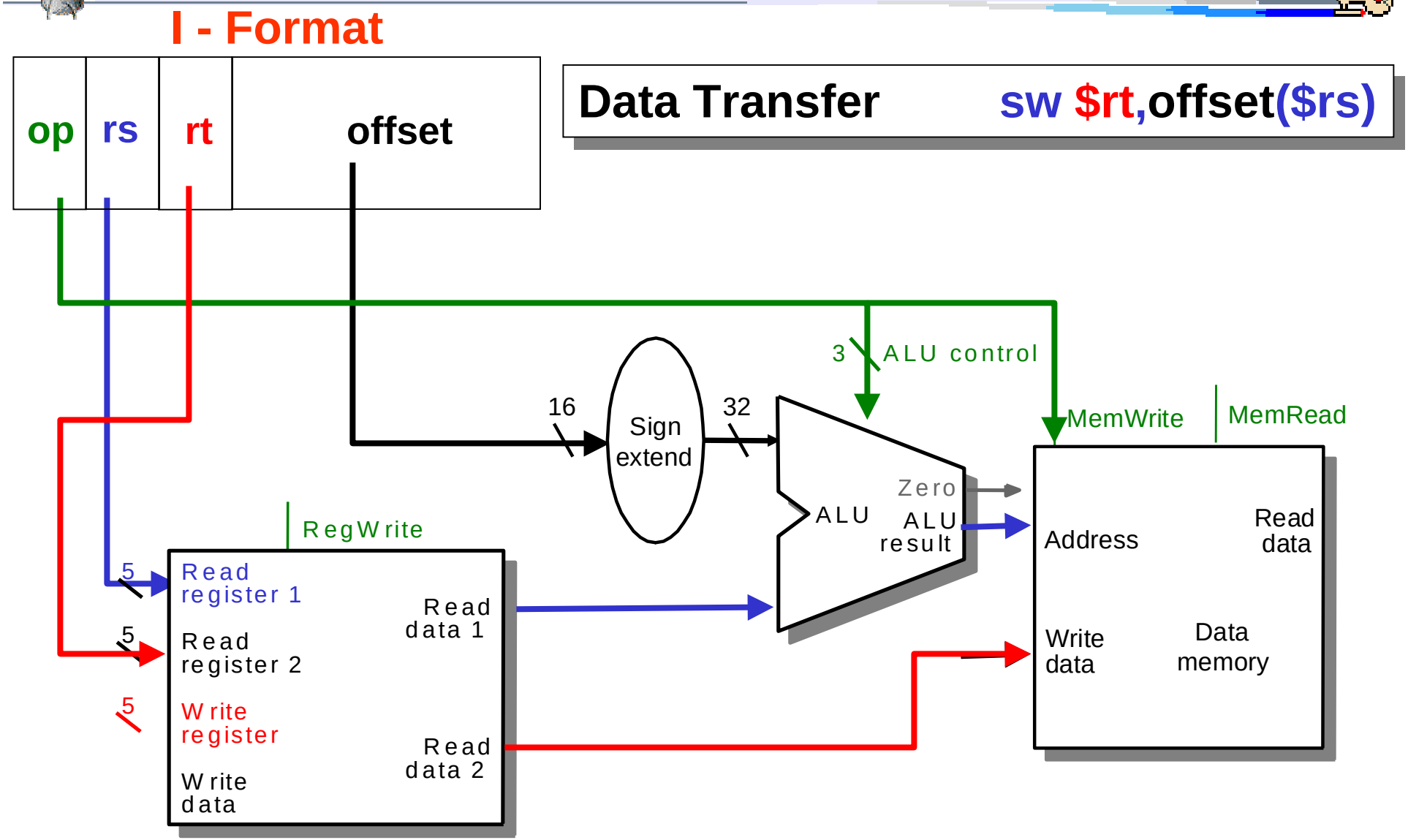
Review: R-type instruction datapath (chapter 5)



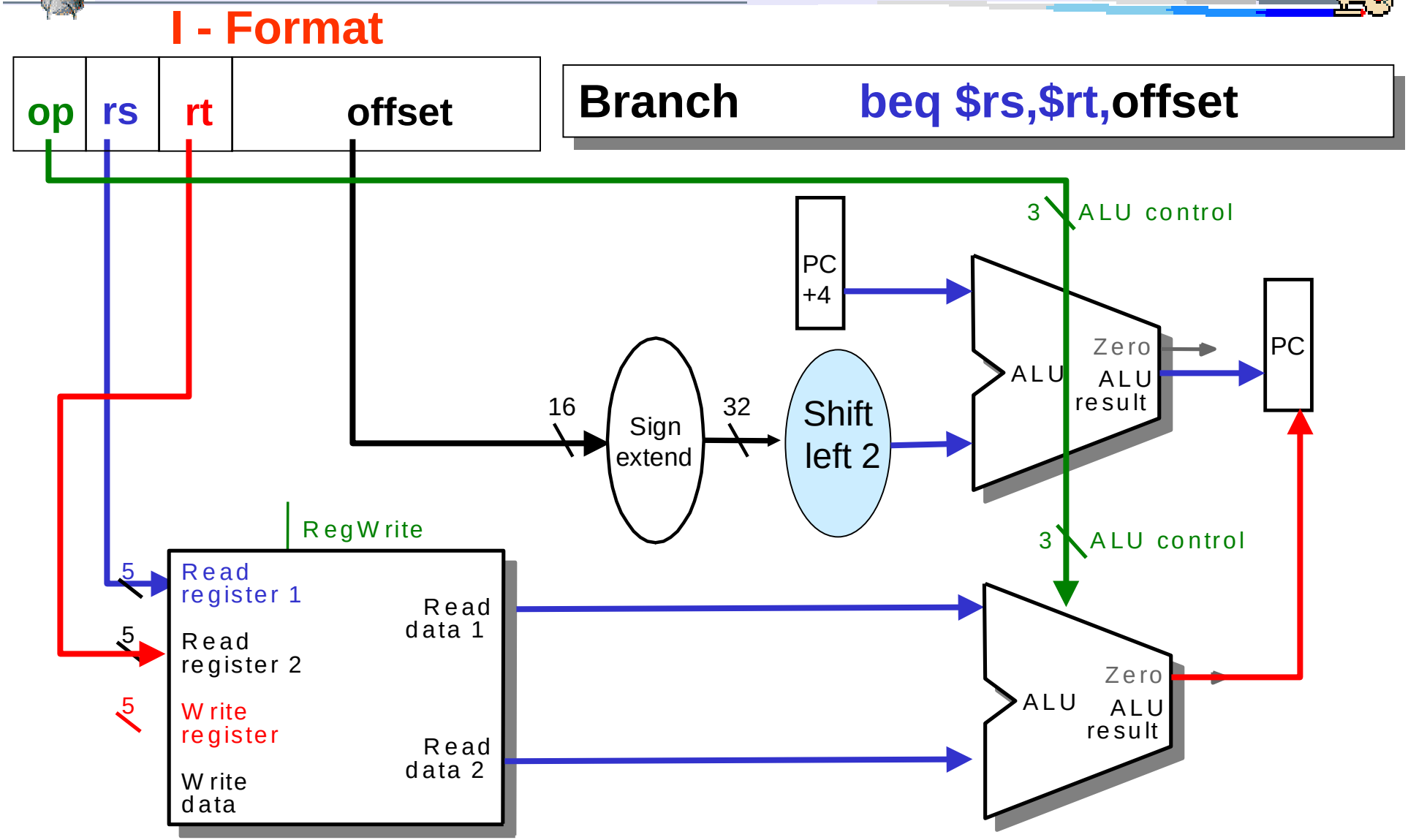
Review: Lw I-type instruction datapath



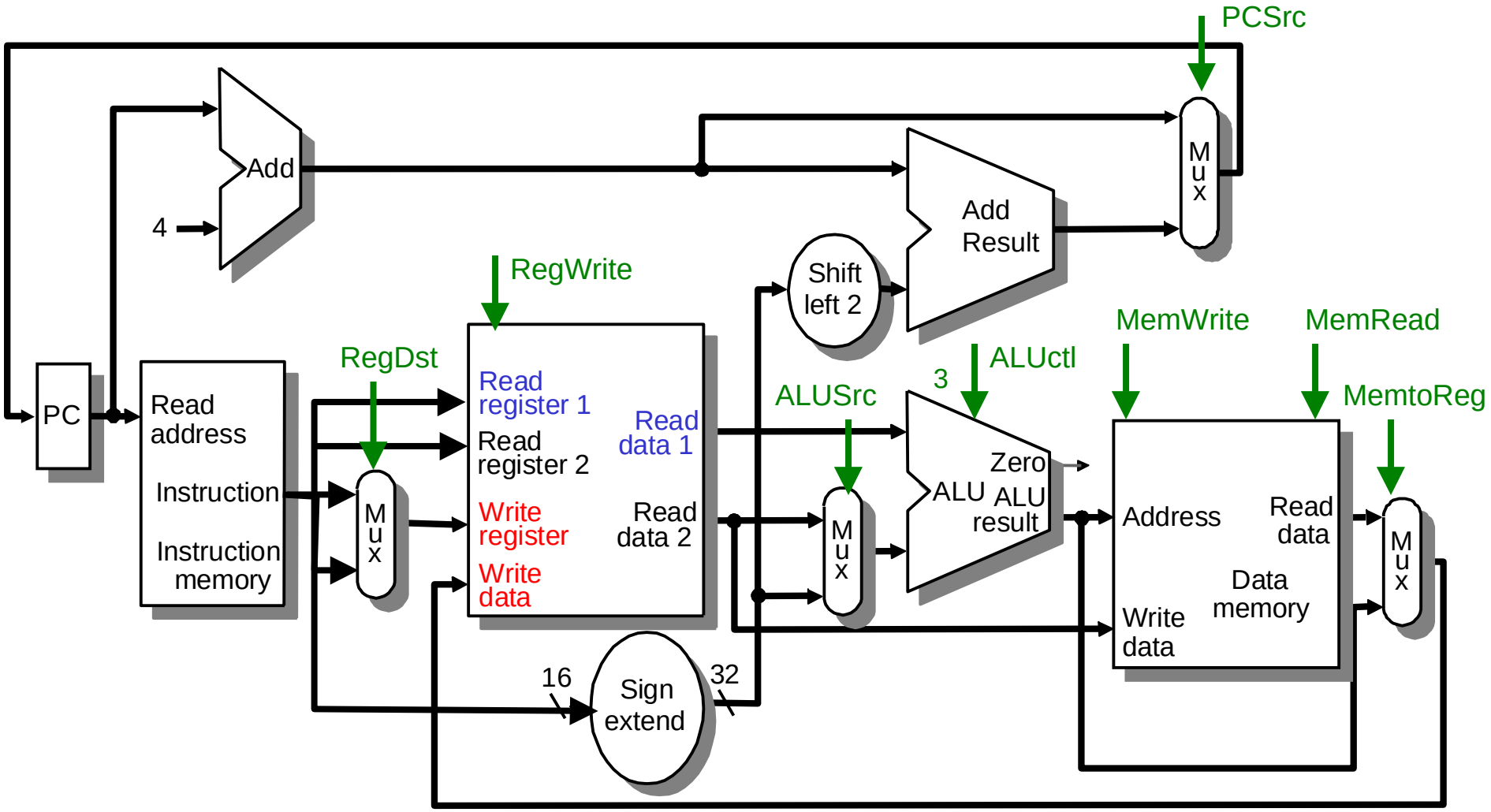
Review: Sw I-type instruction datapath



Review: Branch I-type instruction datapath



Review: Single-cycle processor architecture



ALU decoder (Figures 5.14-15)

Machine opcode	Instruct Format	IR[31-26] Opcode →	ALUop	IR[5-0] Funct →	ALUctl	ALUctl
lw	I-type	100011	00	XXXXXX	add	010
sw	I-type	101011	00	XXXXXX	add	010
beq	I-type	000100	01	XXXXXX	sub	110
add	R-type	000000	10	100000	add	010
sub	R-type	000000	10	100010	sub	110
and	R-type	000000	10	100100	and	000
or	R-type	000000	10	100101	or	001
slt	R-type	000000	10	101010	slt	111

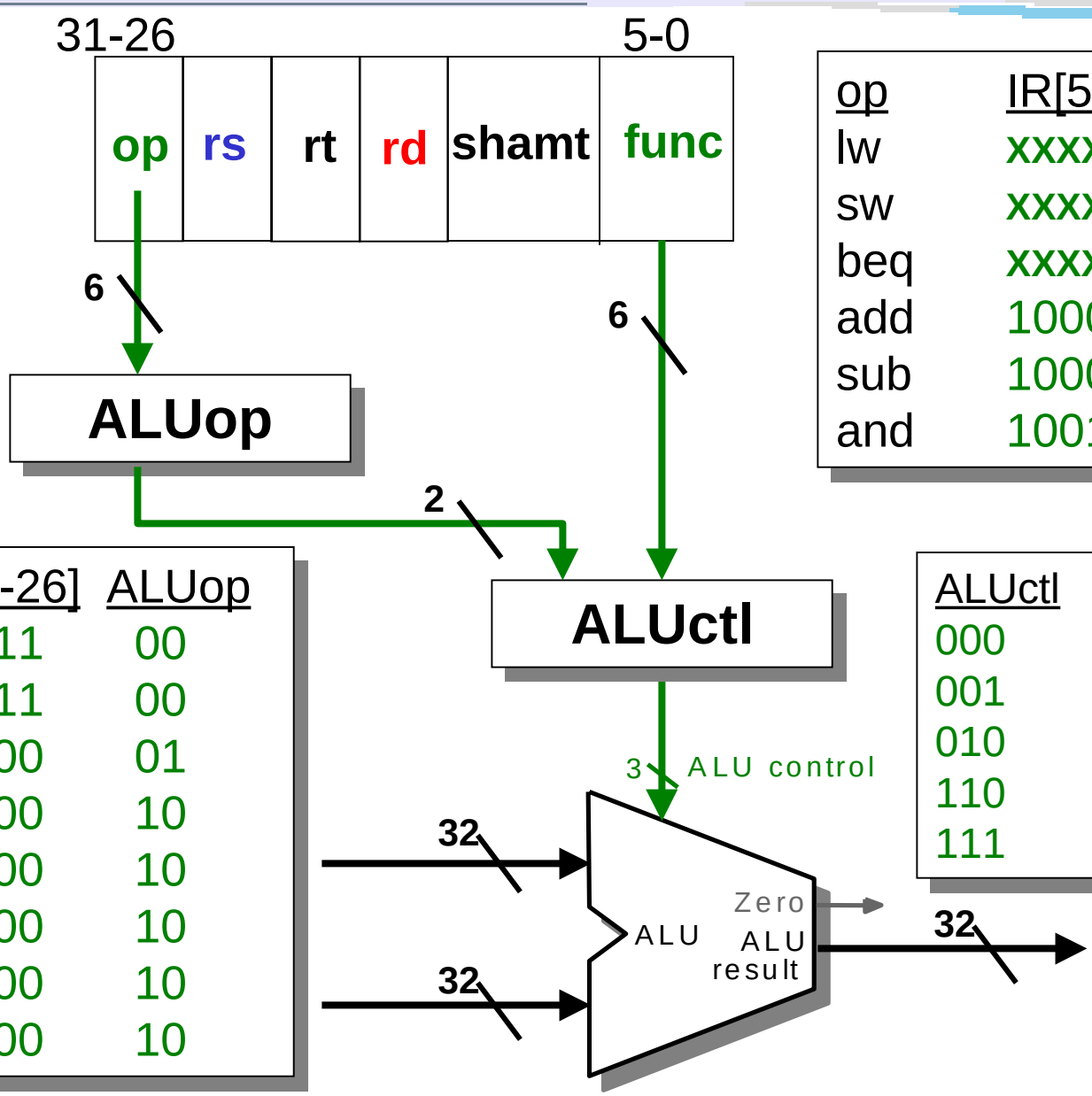
ALUop = ALUopDecoder(Opcode);

ALUctl = ALUctlDecoder(ALUop, Funct);

Note: the Opcode field determines the I-type and R-type

Note: the Funct field determines the ALUctl for R-type

ALU decoders: ALUop and ALUctl

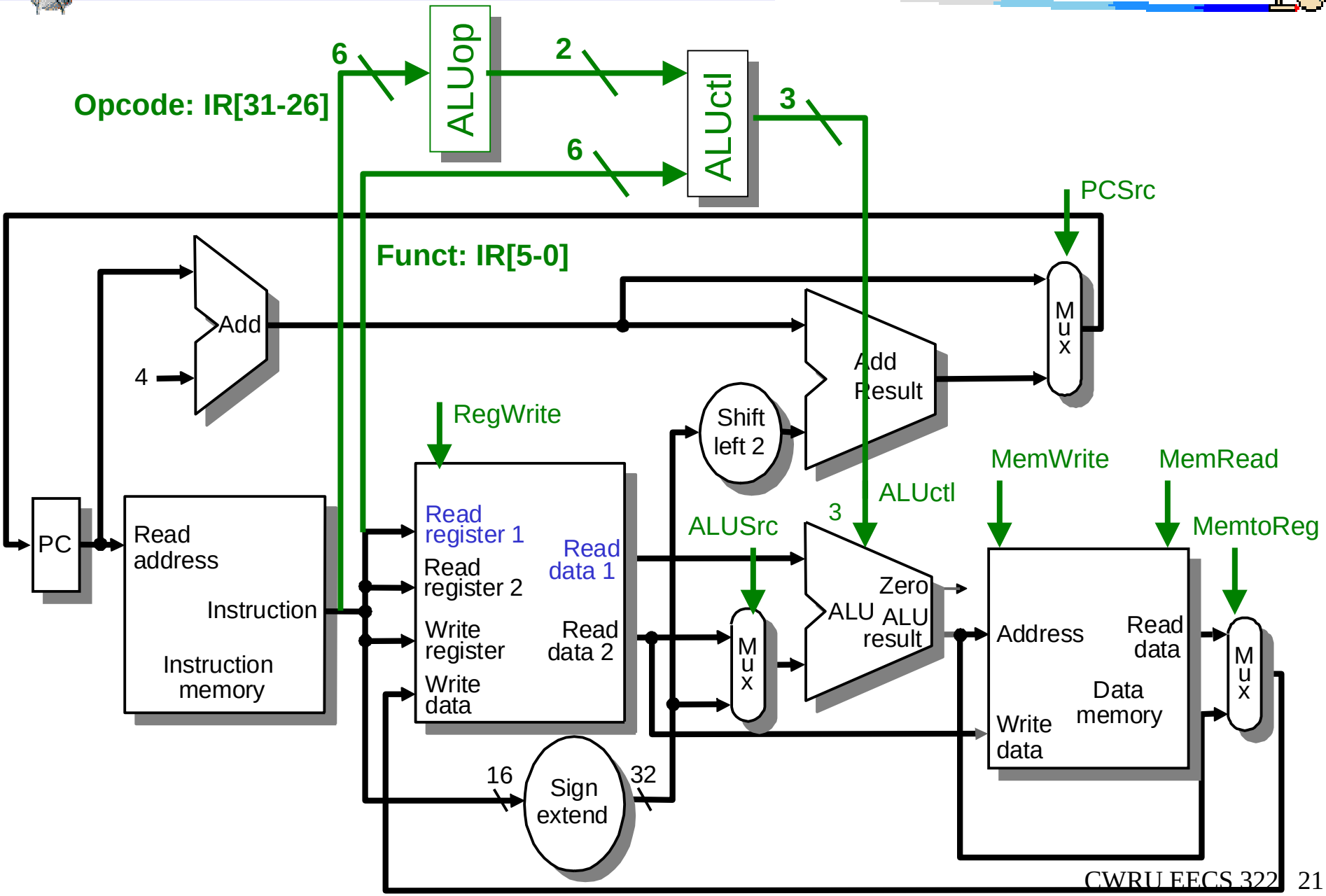


op	IR[5-0]
lw	xxxxxx
sw	xxxxxx
beq	xxxxxx
add	100000
sub	100010
and	100100

op	IR[31-26]	ALUop
lw	100011	00
sw	101011	00
beq	000100	01
add	000000	10
sub	000000	10
and	000000	10
or	000000	10
slt	000000	10

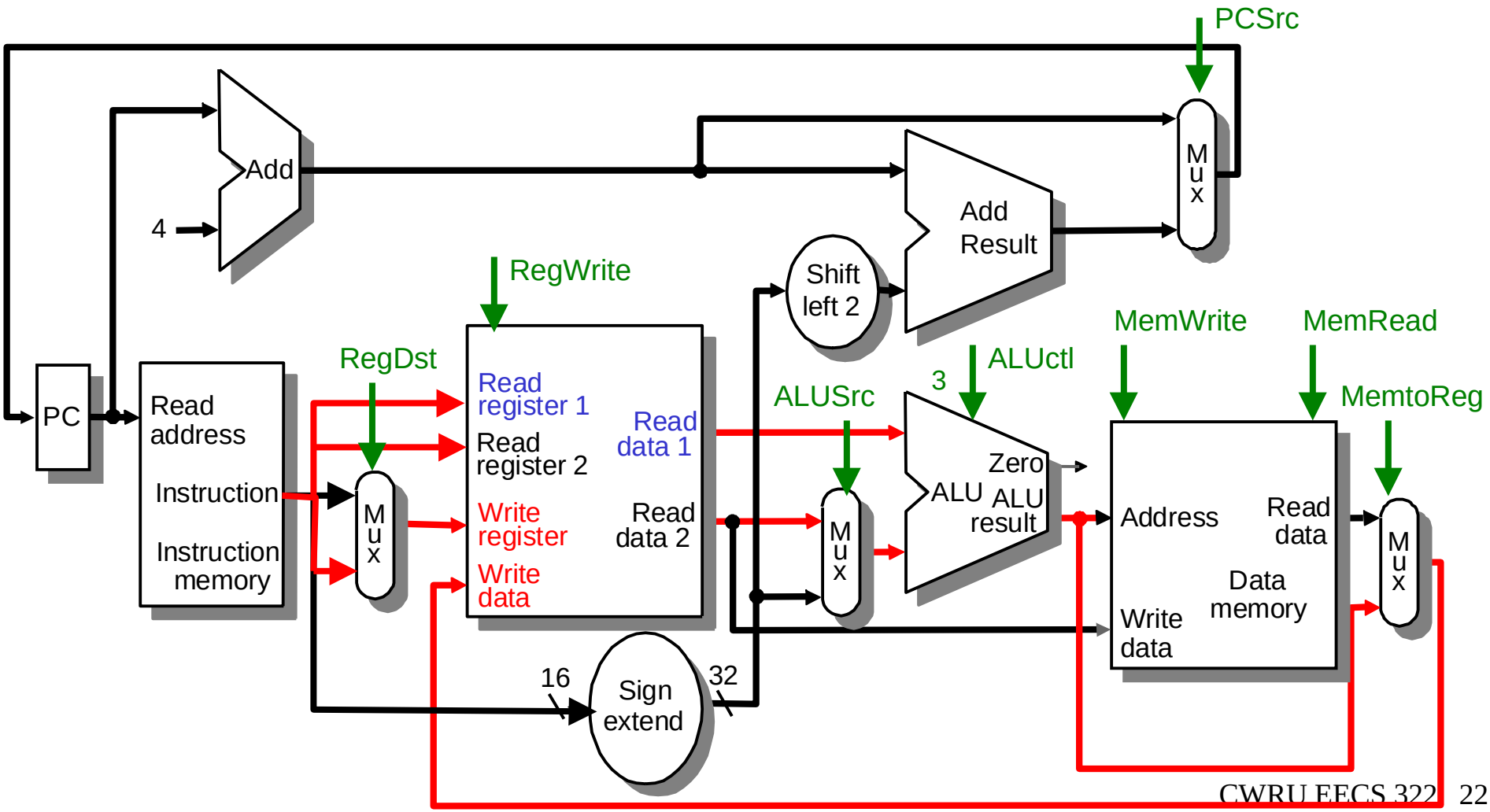
ALUctl	Function
000	bitwise and
001	bitwise or
010	integer add
110	integer sub
111	set less than

Processor architecture with ALU decoder



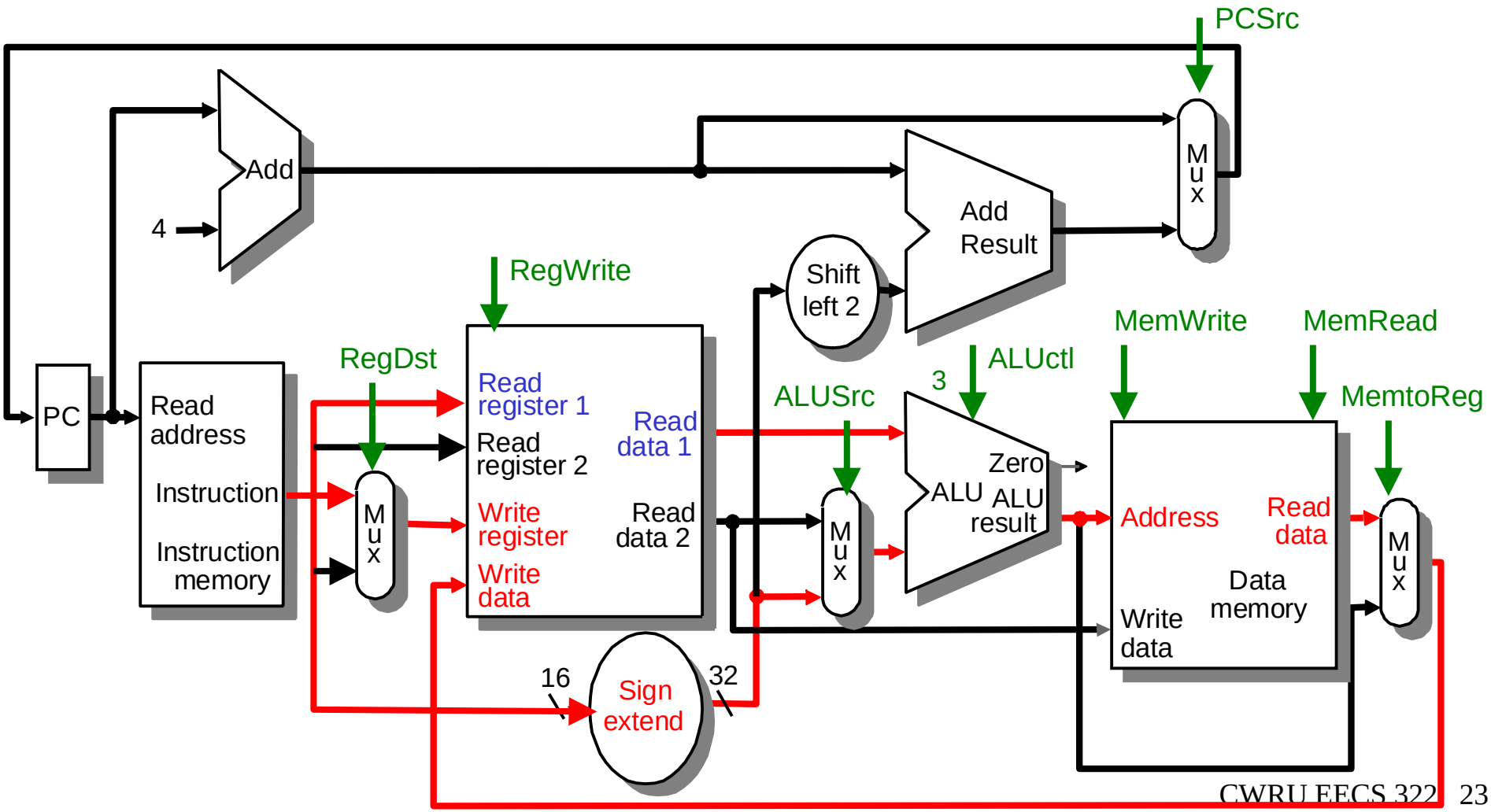
R-format datapath control (Figures 5.20-24)

Machine opcode	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop
R-format	1 ($rd)	0 ($rt)	0(alu)	1	0	0	0	10 (func)



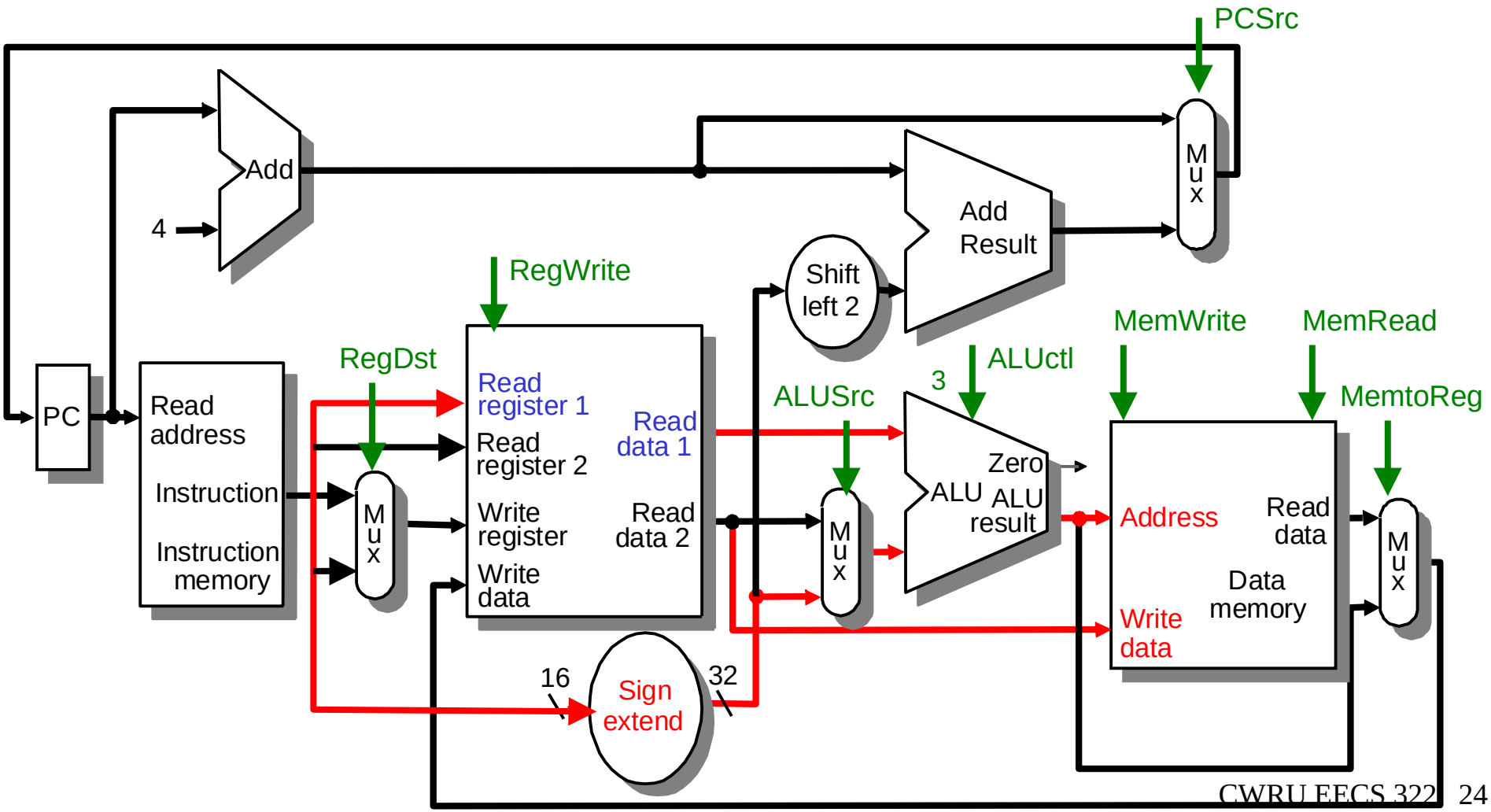
lw datapath control (Figure 5.25)

Machine opcode	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop
lw	0 ($rt)	1 (offset)	1(mem)	1	1	0	0	01 (add)



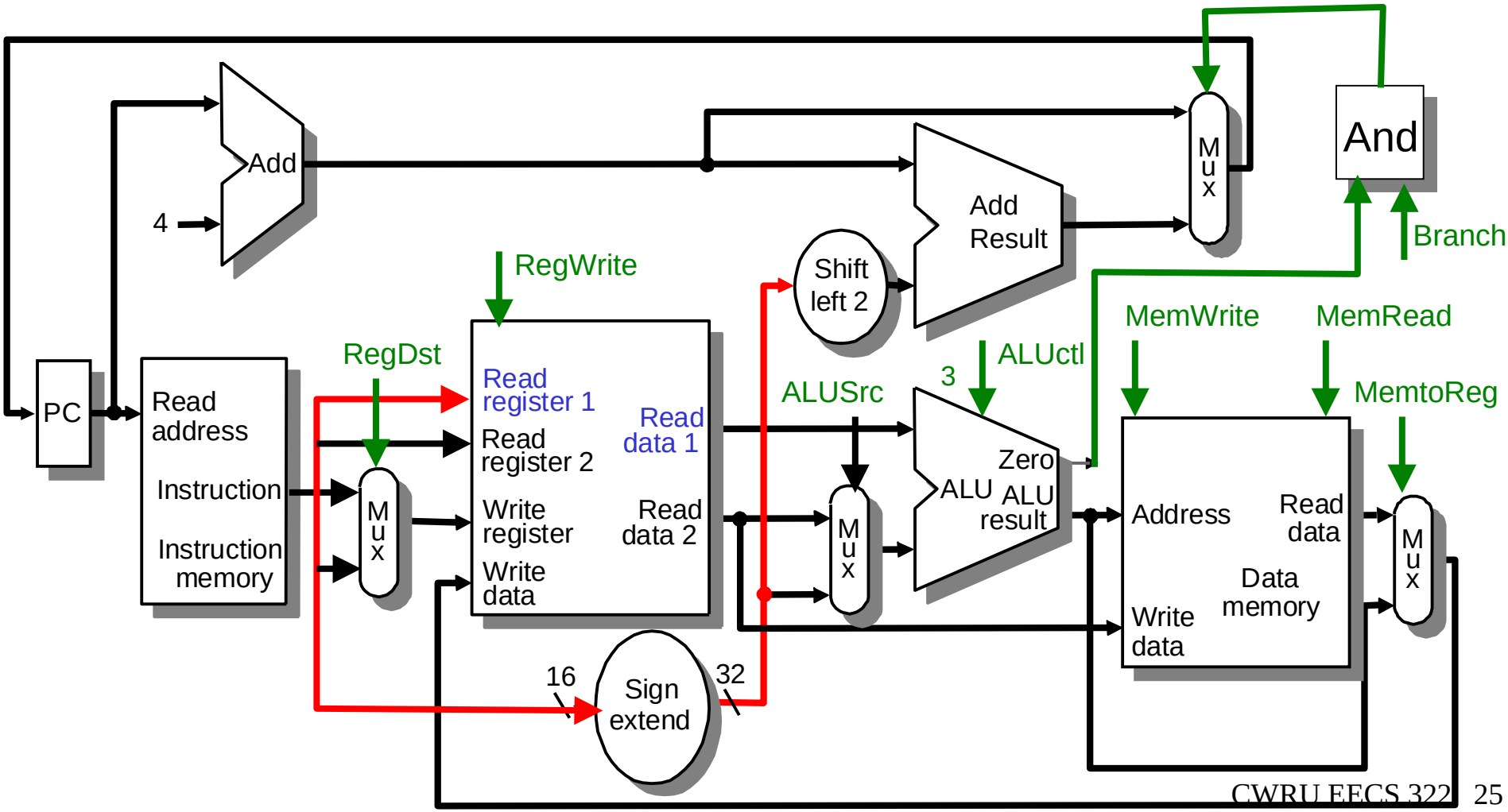
sw datapath control

Machine opcode	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop
sw	X	1 (offset)	X	0	0	1	0	01 (add)



beq datapath control (Figure 5.26)

Machine opcode	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop
beq	X	0	X	0	0	0	1	01 (sub)



Single/Multi-Clock Comparison

(page 373-5)

A multi-cycle processor has the following instruction times

add(44%)= 6ns = Fetch(2ns) + RegR(1ns) + ALU(2ns) + RegW(1ns)

lw(24%) =8ns = Fetch(2ns) +RegR(1ns) +ALU(2ns) +MemR(2ns)+RegW(1ns)

sw(12%) = 7ns = Fetch(2ns) + RegR(1ns) + ALU(2ns) + MemW(2ns)

beq(18%)= 5ns = Fetch(2ns) + RegR(1ns) + ALU(2ns)

j(2%) = 2ns = Fetch(2ns)

Single-cycle CPI = 44%×8ns +24%×8ns +12%×8ns +18%×8ns +2%×8ns = 8ns

Multi-cycle CPI = 44%×6ns +24%×8ns +12%×7ns +18%×5ns +2%×2ns = 6.3ns

$$\frac{\text{single}-\text{cycle CPI}}{\text{multi}-\text{cycle CPI}} = \frac{8\text{ns}}{6.3\text{ns}} = 1.27 \text{ times faster}$$

Architectural improved performance without speeding up the clock!

Single-cycle problems

- Single Cycle Problems:
  - Clock cycle is the slowest instruction delay = 8ns = 125MHz
  - What if we had a more complicated instruction like floating point? (fadd = 30ns, fmul=100ns) Then clock cycle = 100ns = 10 Mhz
  - Wasteful of chip area (2 adders + 1 ALU). Cannot reuse resources.
  - Wasteful of memory: separate instructions & data (Harvard architecture)

- Solutions:
  - Use a “smaller” cycle time (if the technology can do it)
  - Have different instructions take different numbers of cycles (multi-cycle)
  - Better reuse of functional units: a “multicycle” datapath (1 ALU instead of 3 adders)

- Multi-cycle approach
  - Clock cycle is the slowest function unit = 2ns = 500MHz
  - We will be reusing functional units:
    ALU used to increment PC (Adder1) & compute address (Adder2)
  - Memory reused for instruction and data (Von Neuman architecture)

Some Design Trade-offs

<u>High level design techniques</u>

Algorithms: change instruction usage

minimize $\sum n_{instruction} * t_{instruction}$

Architecture: Datapath, FSM, Microprogramming

adders: ripple versus carry lookahead

multiplier types, …

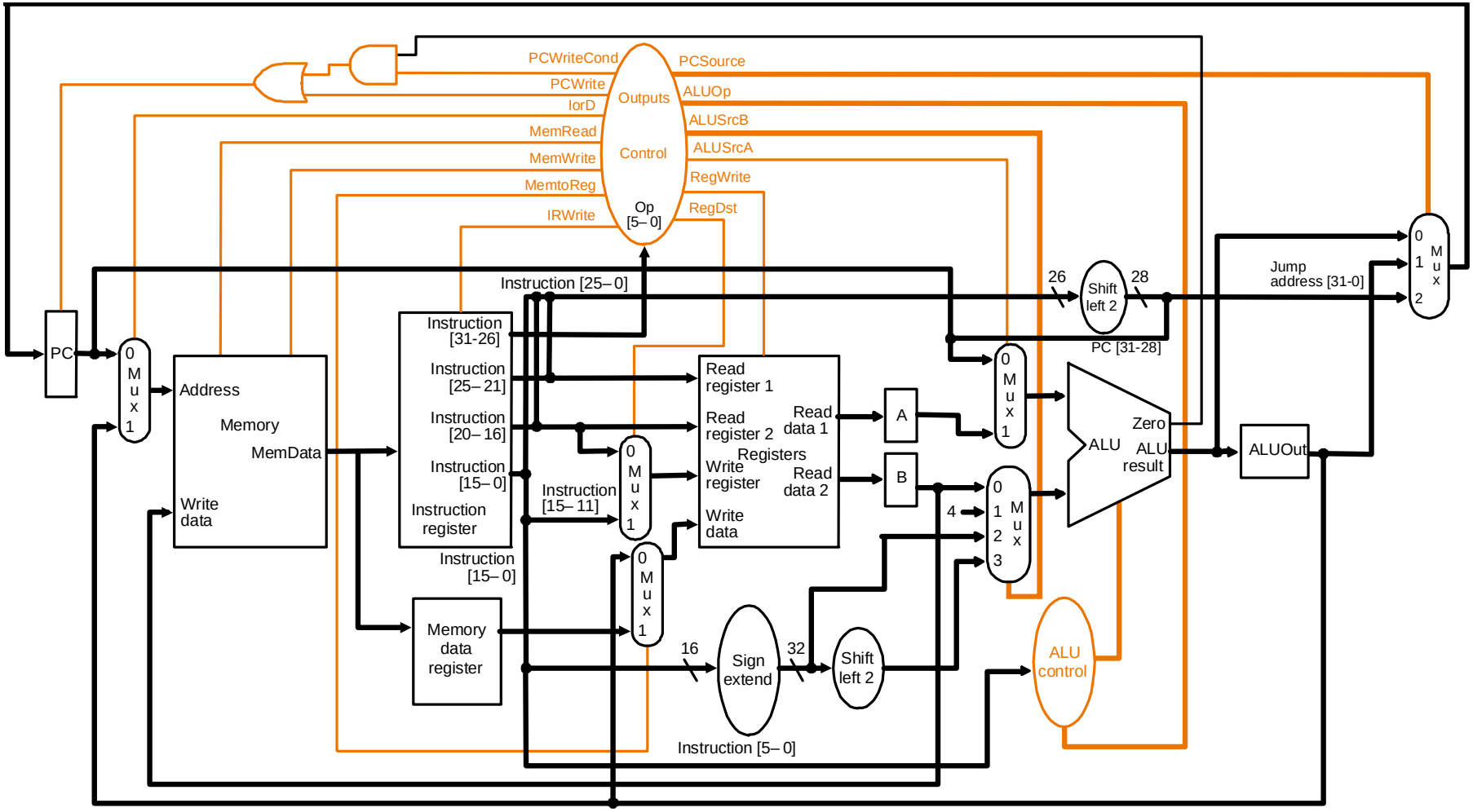
<u>Lower level design techniques (closer to physical design)</u>

clocking: single verus multi clock

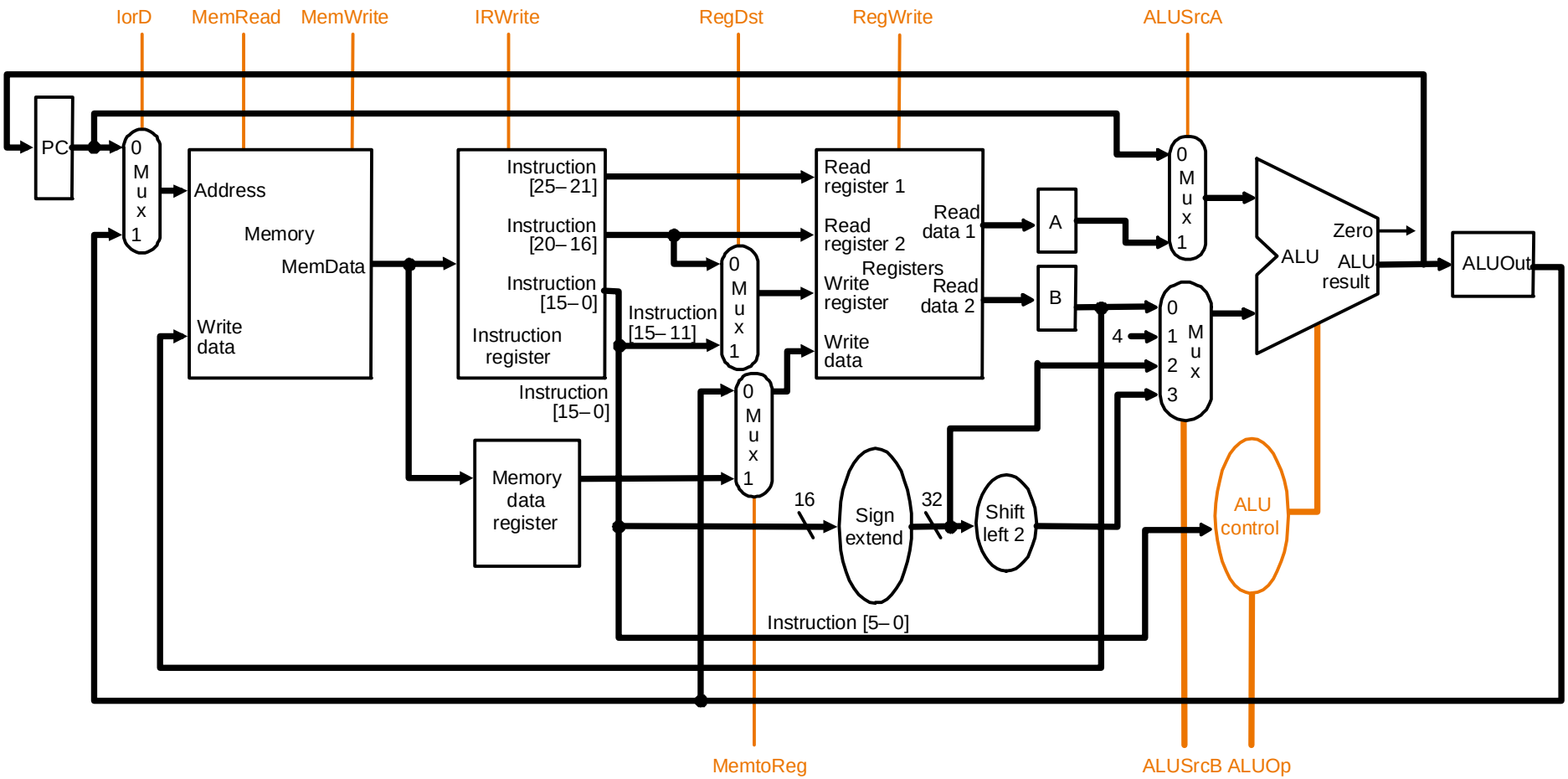
technology: layout tools: better place and route

process technology: 0.5 micron to .18 micron

Multi-cycle Datapath: with controller



Multi-cycle Datapath



Multi-cycle = 1 Mem + 5.5 Muxes + 1 ALU + 5 Registers (A,B,IR,MDR,ALUOut)

Single-cycle = 2 Mem + 4.0 Muxes + 1 ALU + 2 adders

Multi-cycle: 5 execution steps

- T_1 (a,lw,sw,beq,j) Instruction Fetch
- T_2 (a,lw,sw,beq,j) Instruction Decode
  and Register Fetch
- T_3 (a,lw,sw,beq,j) Execution, Memory Address Calculation,
  or Branch Completion
- T_4 (a,lw,sw) Memory Access
  or R-type instruction completion
- T_5 (a,lw) Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Multi-cycle Approach

All operations in each clock cycle T_i are done in parallel not sequential!

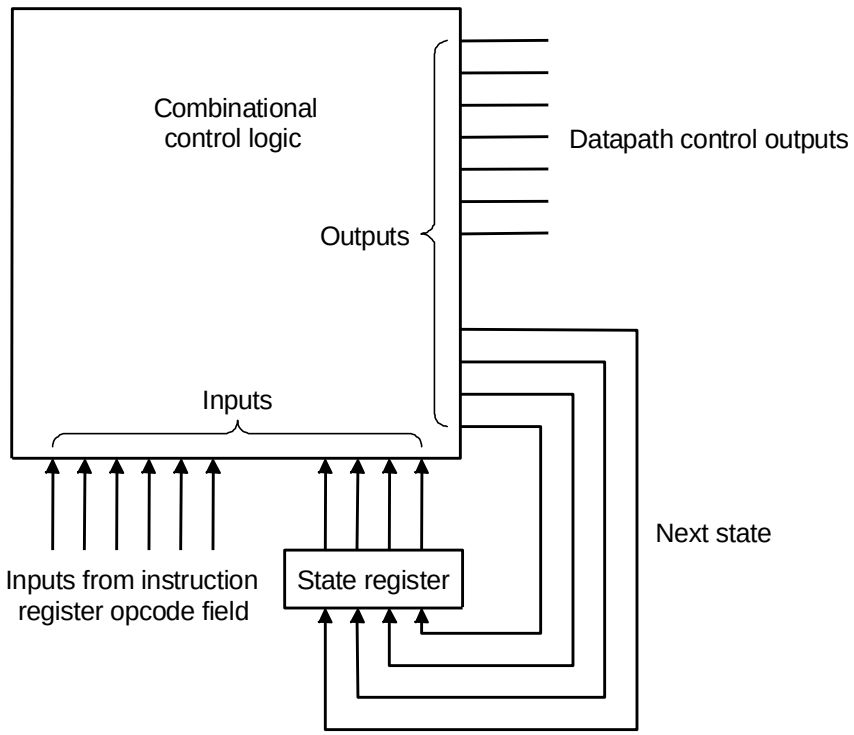
For example, T_1, IR = Memory[PC] and PC=PC+4 are done simultaneously!

	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
T_1	Instruction fetch	IR = Memory[PC] PC = PC + 4			
T_2	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
T_3	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
T_4	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
T_5	Memory read completion		Load: Reg[IR[20-16]] = MDR		

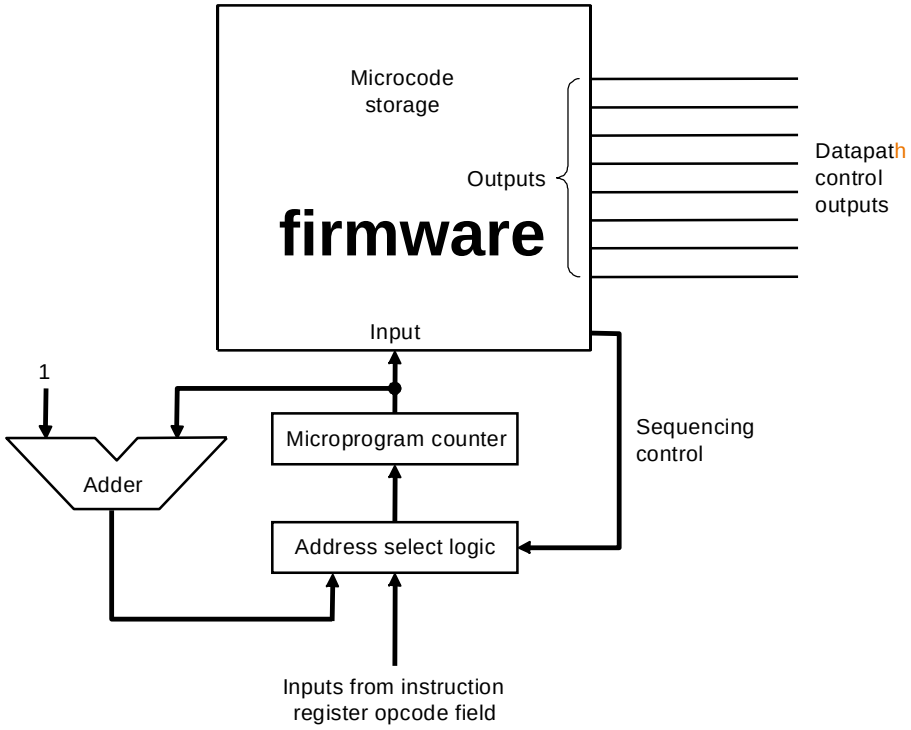
Between Clock T_2 and T_3 the microcode sequencer will do a dispatch 1

Multi-cycle using Microprogramming

Finite State Machine
(*hardwired control*)



Microcode controller



Requires microcode memory to be faster than main memory

Microcode: Trade-offs

- Distinction between specification & implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write (maintenance)
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers
- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Microinstruction format: Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!

- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower

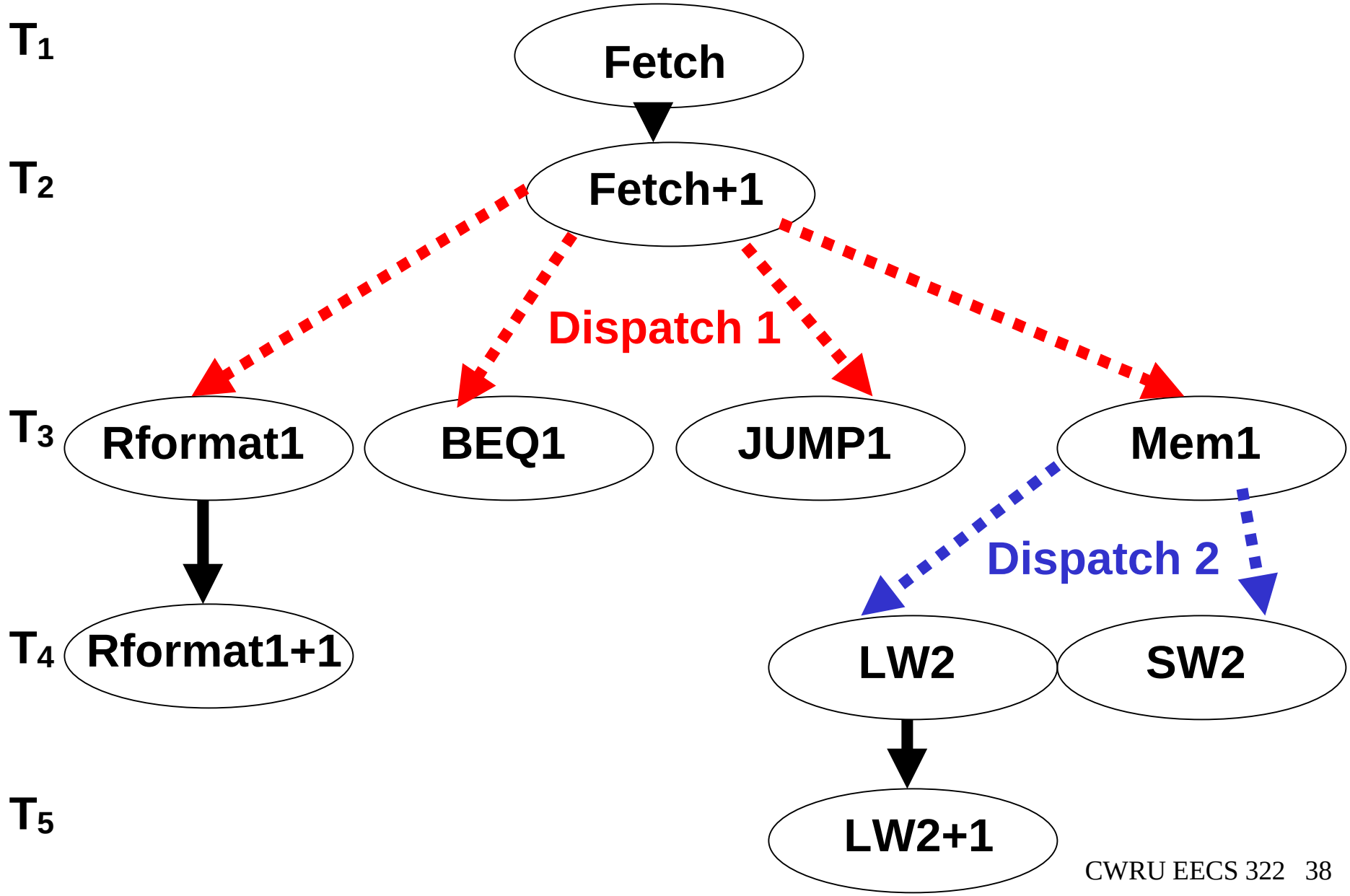
- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It's easy to add new instructions

Microprogramming: program

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

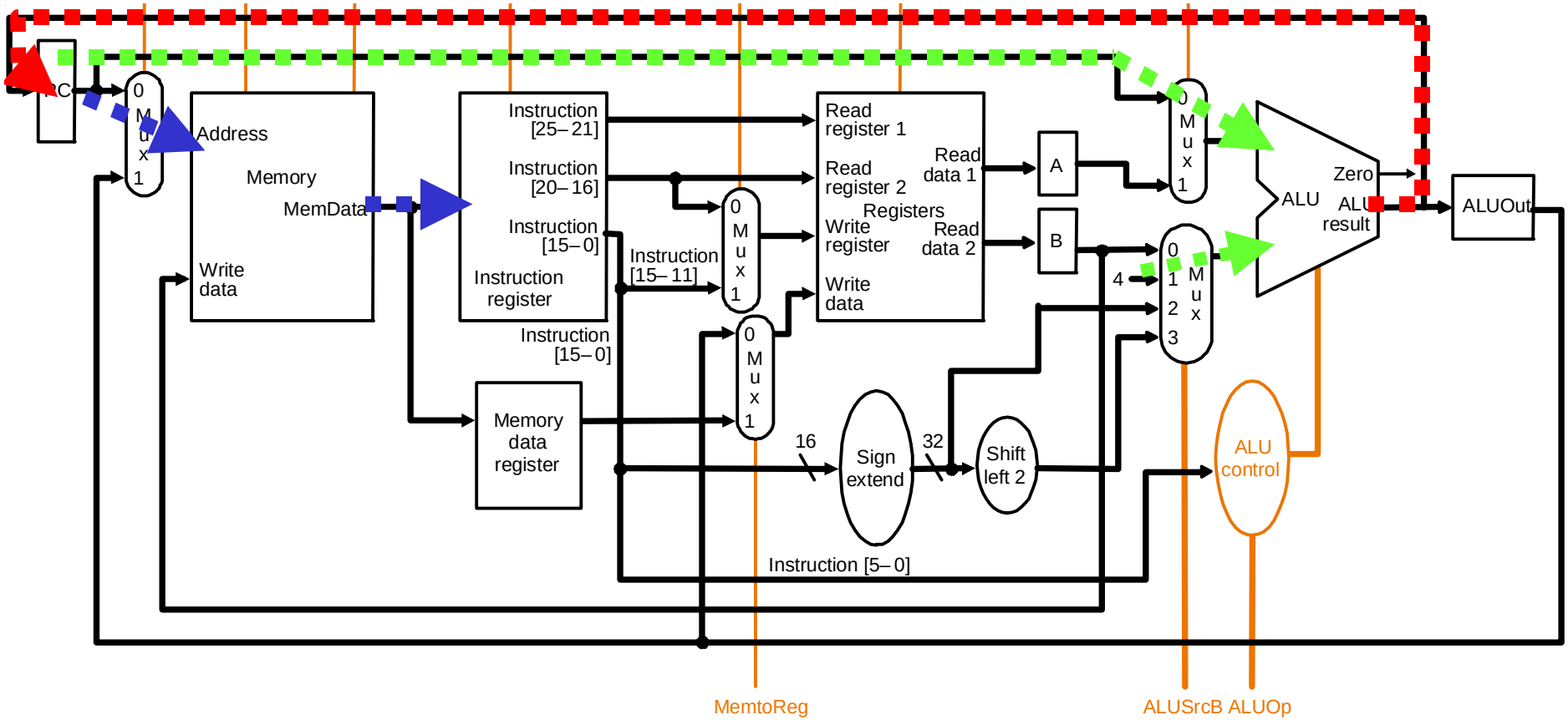
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Microprogramming: program overview



Microprogram steping: T_1 Fetch

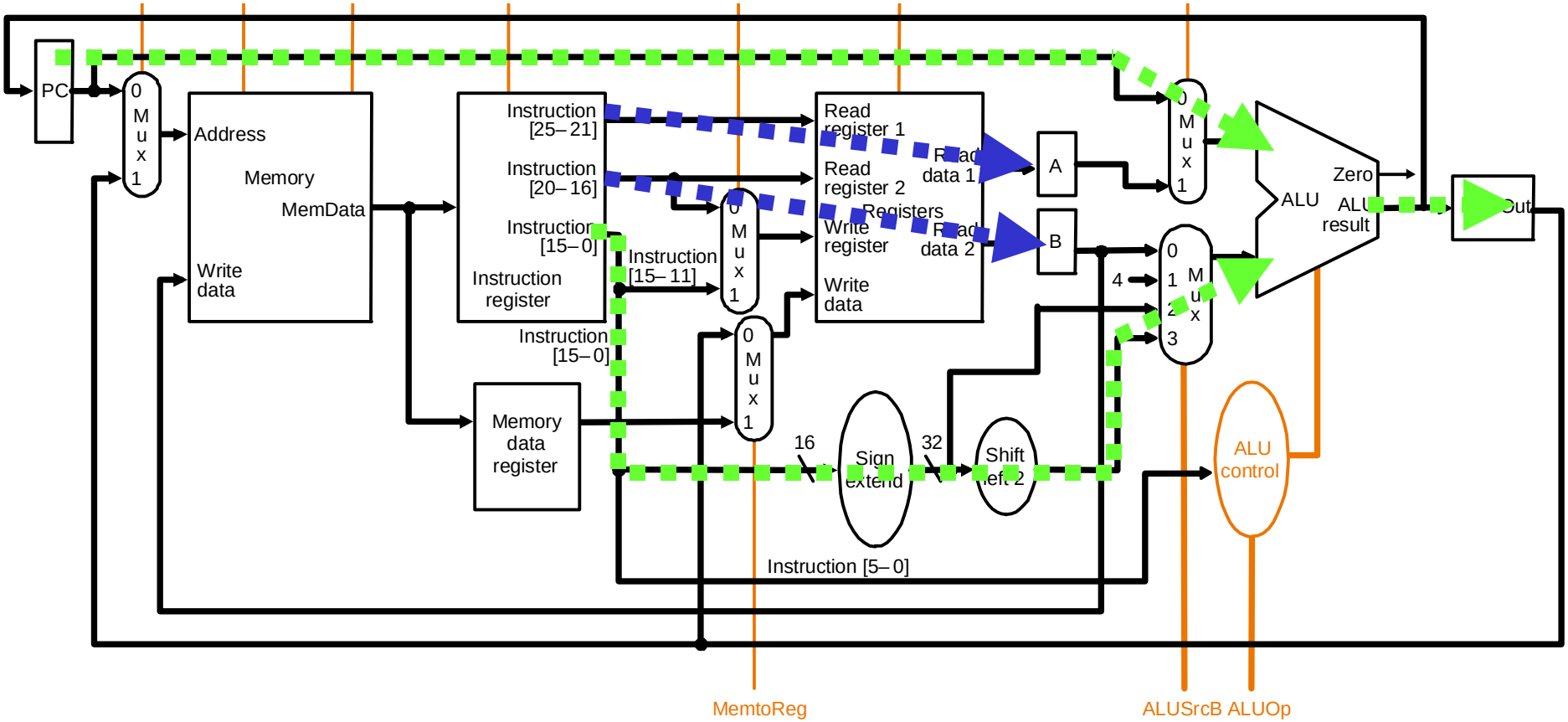
(Done in parallel) IR←MEMORY[PC] & PC ← PC + 4



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Fetch	add	pc	4		ReadPC	ALU	Seq

T_2 Fetch + 1

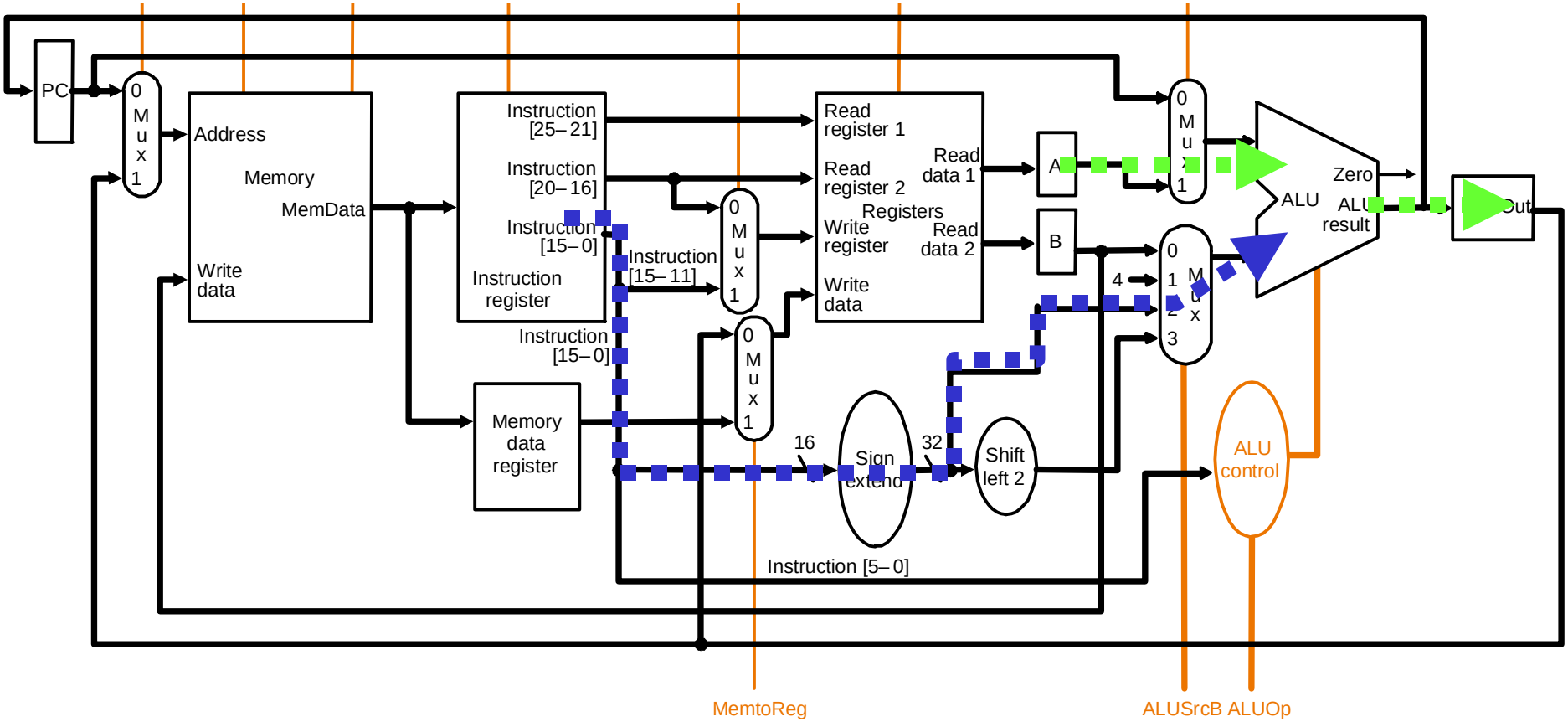
A←Reg[IR[25-21]] & B←Reg[IR[20-16]] & ALUOut←PC+signext(IR[15-0]) <<2



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
	add	pc	ExtSh	Read			D#1

T_3 Dispatch 1: Mem1

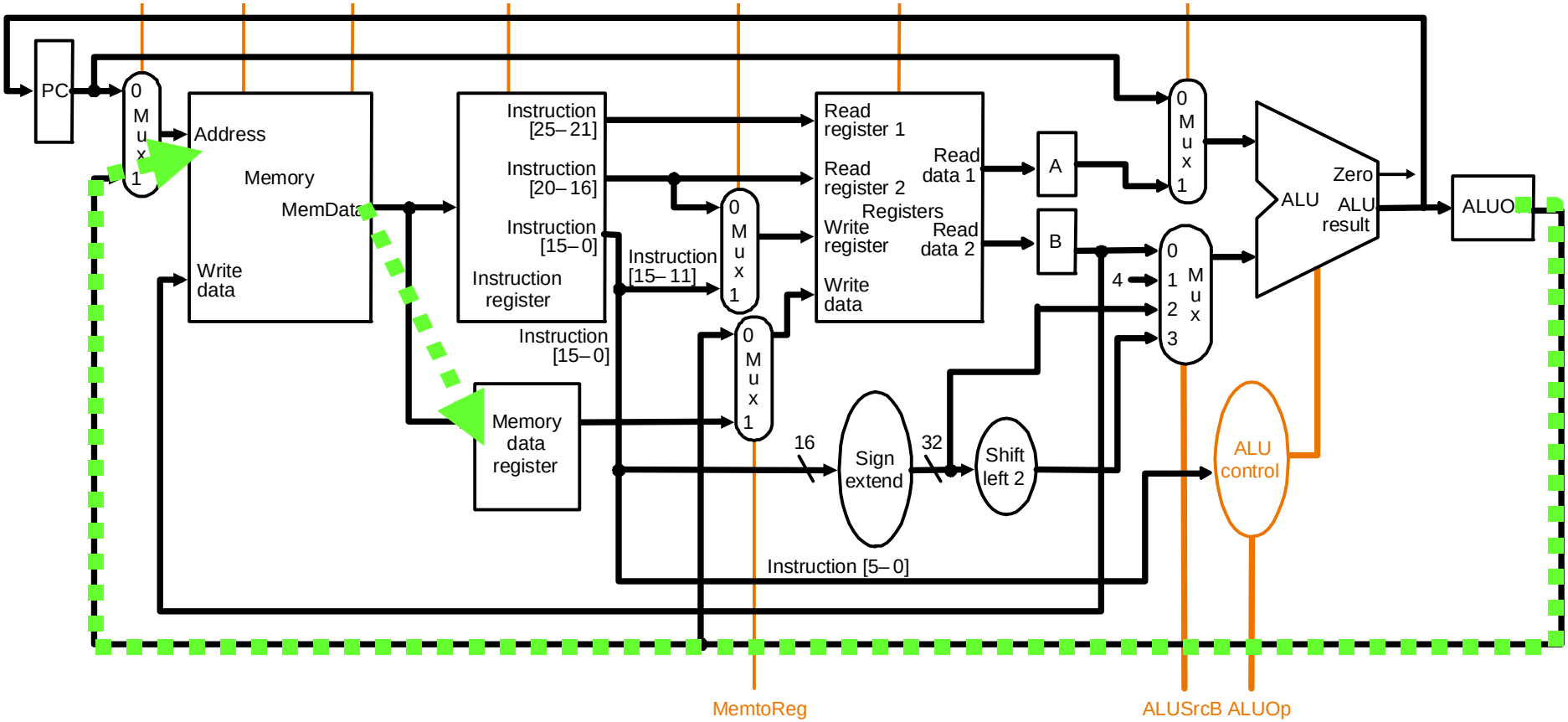
ALUOut ← A + sign_extend(IR[15-0])



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Mem1	add	A	ExtSh				D#2

T_4 Dispatch 2: LW2

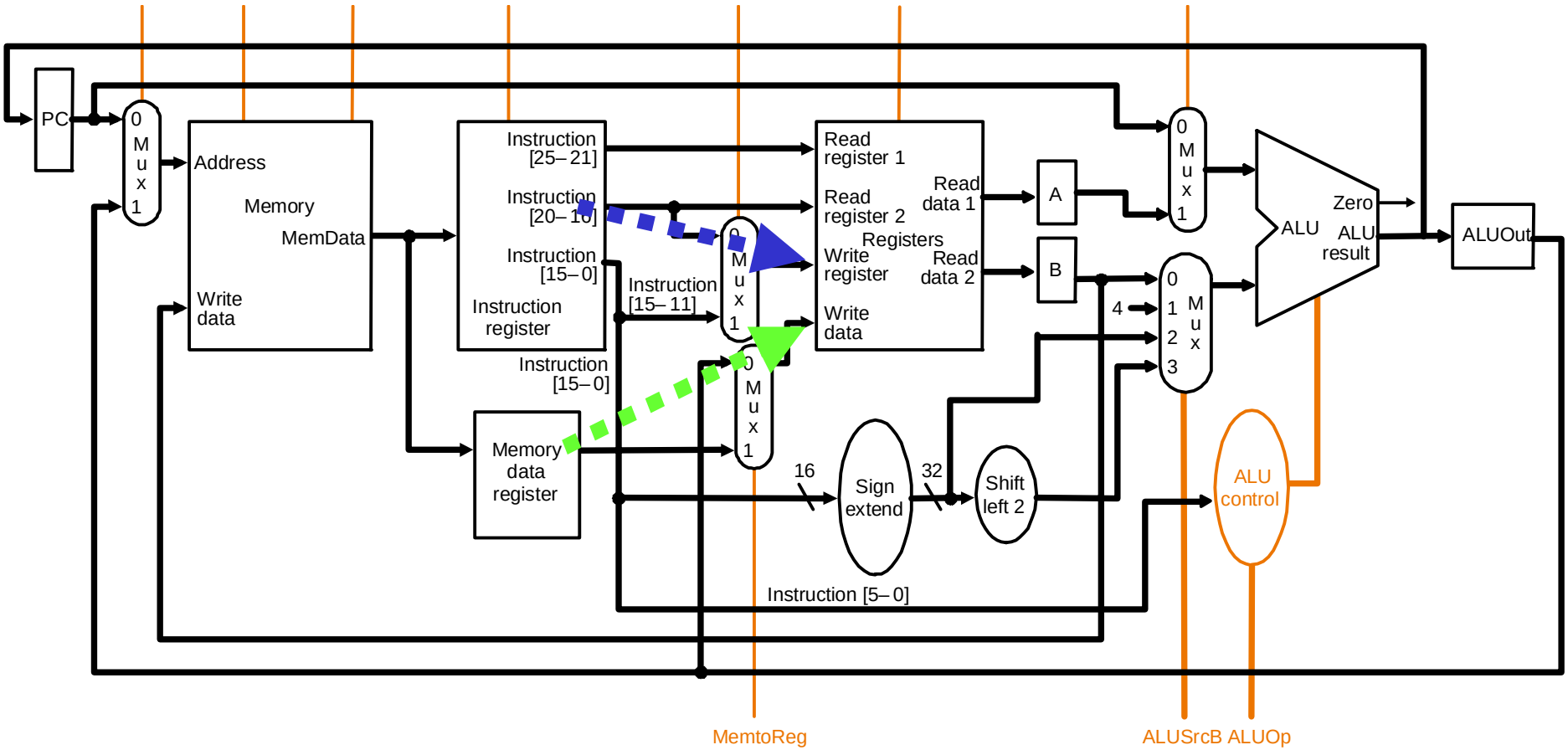
MDR ← Memory[ALUOut]



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
LW2					ReadALU		Seq

T_5 LW2+1

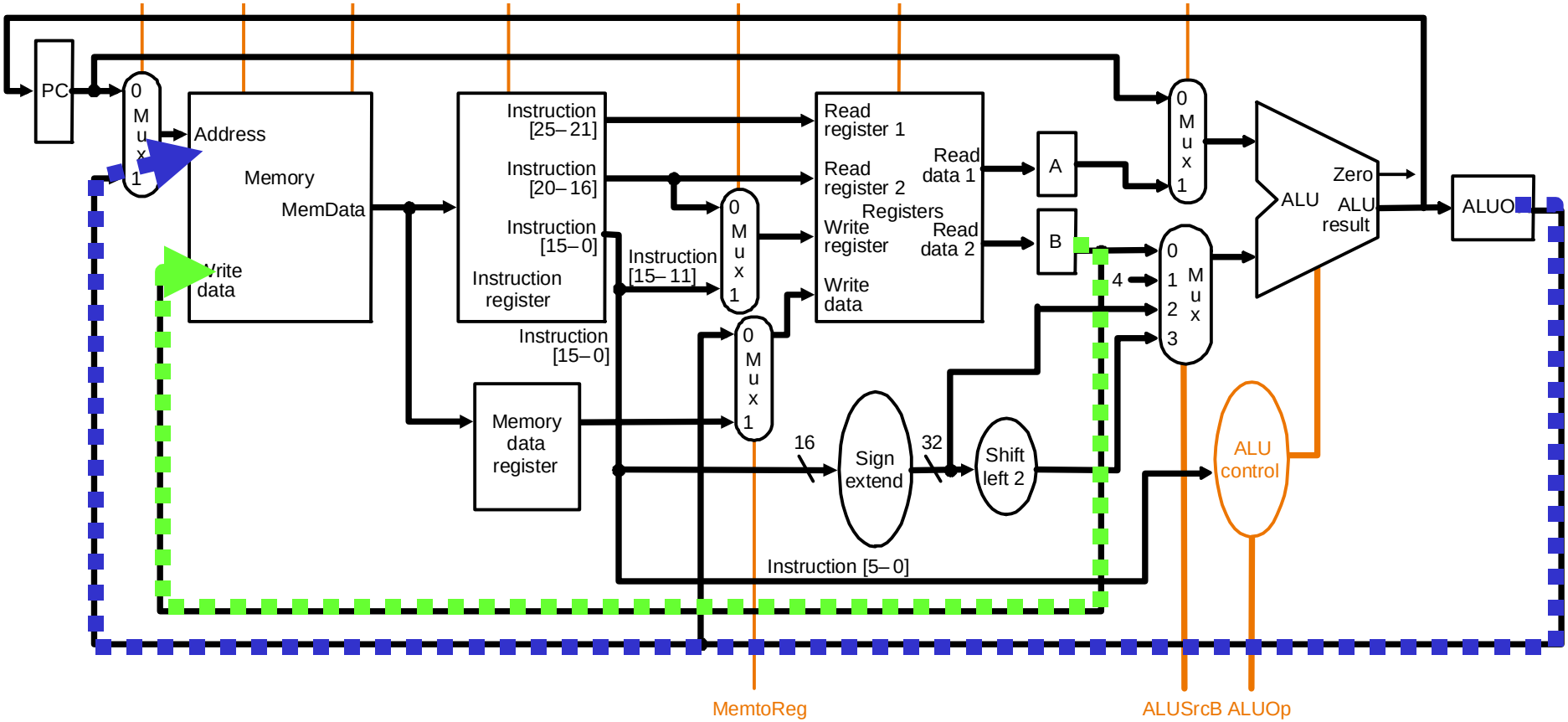
Reg[IR[20-16]] ← MDR



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
					WMDR		Fetch

T_4 Dispatch 2: SW2

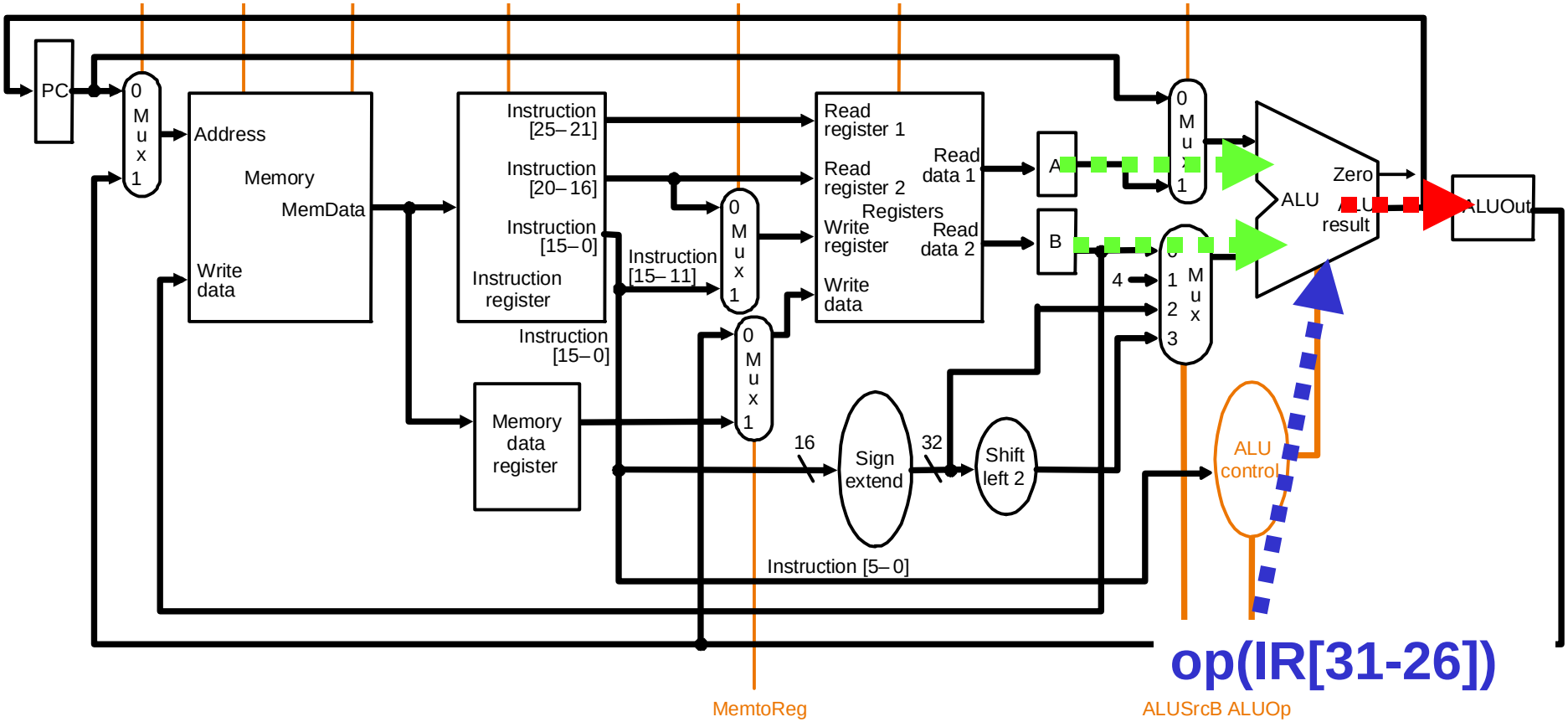
Memory[ALUOut] ← B



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
SW2					WriteALU		Fetch

T_3 Dispatch 1: Rformat1

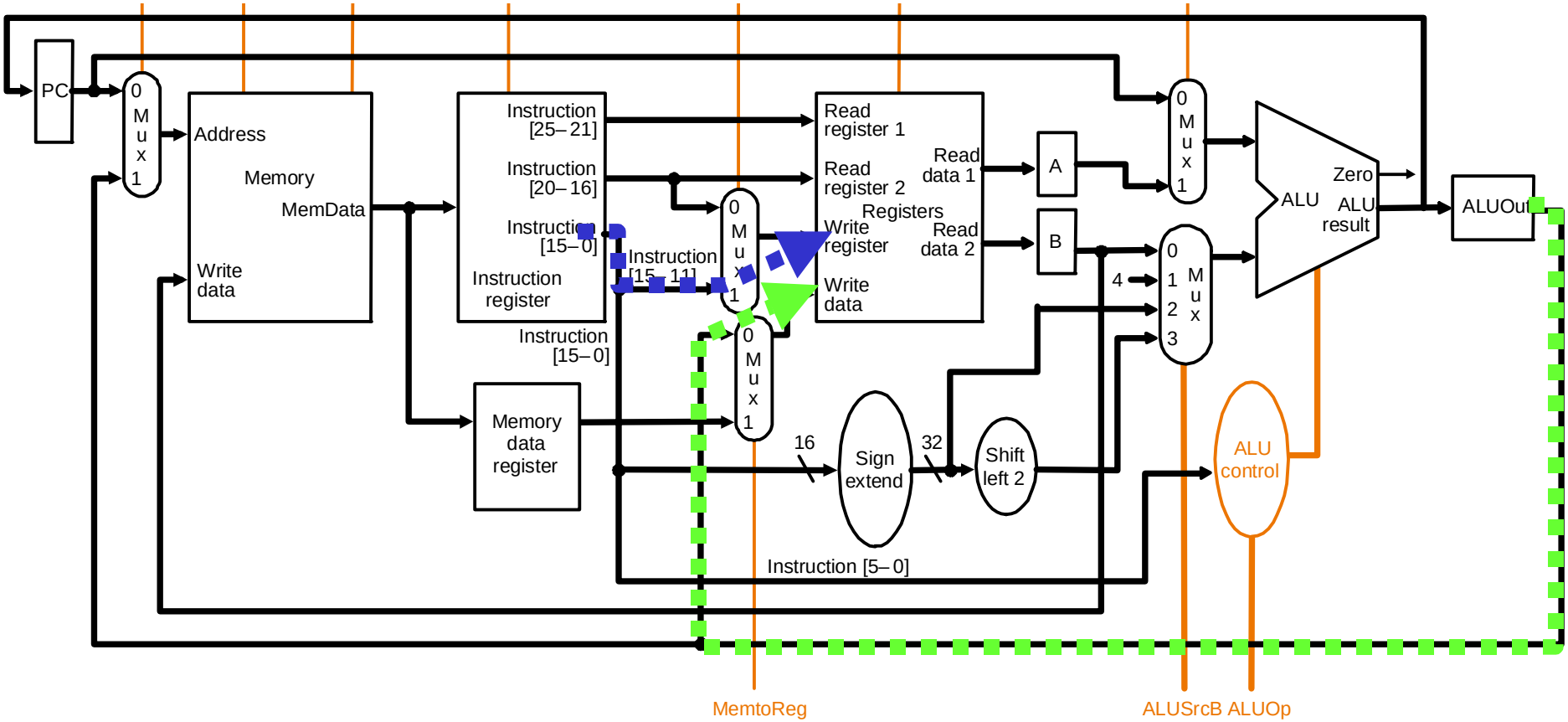
ALUOut ← A op(IR[31-26]) B



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Rf...1	op	A	B				Seq

T_4 Dispatch 1: Rformat1+1

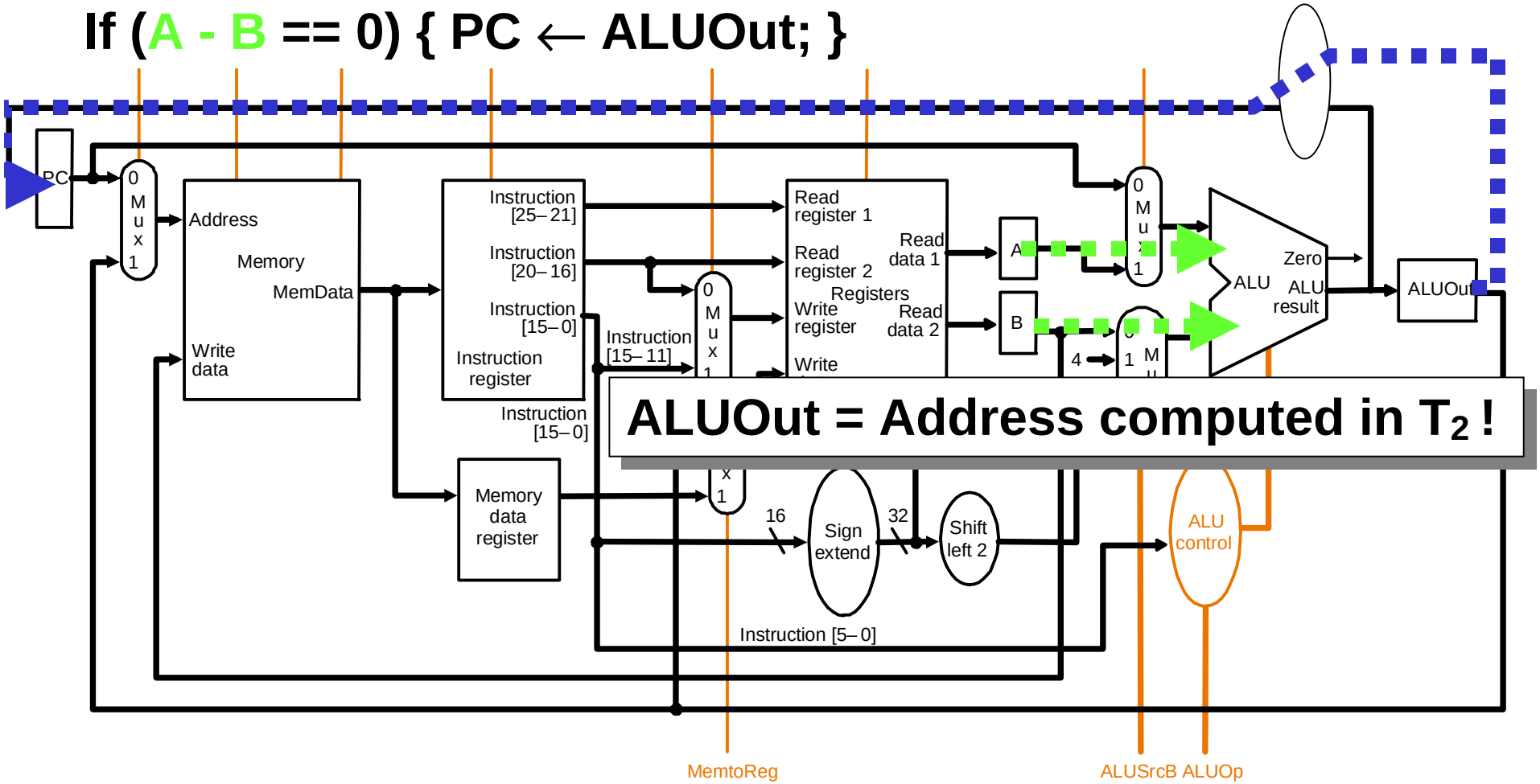
Reg[IR[15-11]] ← ALUOut



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
				WALU			Fetch

T_3 Dispatch 1: BEQ1

If (A - B == 0) { PC ← ALUOut; }

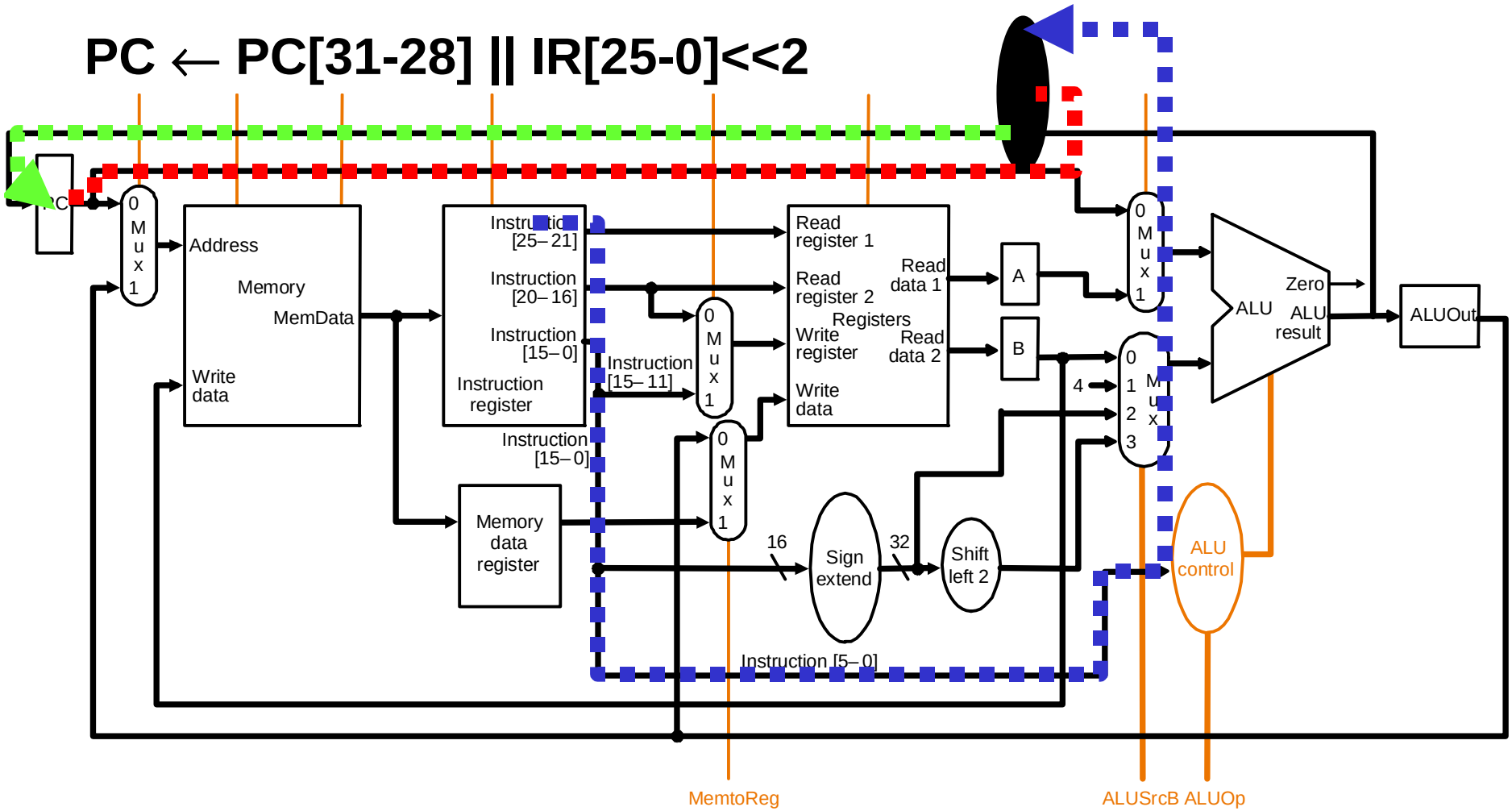


ALUOut = Address computed in T_2 !

Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
BEQ1	subt	A	B			ALUOut-0	Fetch

T_3 Dispatch 1: Jump1

PC ← PC[31-28] || IR[25-0]<<2



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Jump1						Jaddr	Fetch